EE 524 P
Applied High-Performance GPU Computing

LECTURE 6 : Thursday, November 1, 2018

Instructor: Dr. Colin Reinhardt

University of Washington - Professional Masters Program

Autumn 2018

Lecture 6 : Outline

- Final Project Overview
  - Project Proposals: due 11/15 at 6:00 PM
  - Proposal Designs: due 11/29 at 6:00 PM
  - Final Project: due 12/14 by 6:00 PM (<u>no</u> late submissions)
- Parallel Software Patterns
  - Map
  - Shift
  - Geometric Decomposition
  - Stencil
- Working with Images in OpenCL
- Images and Stencil Case Study
- In-class EX6

Final Project: Proposals

DUE: Thursday 11/15 by 6:00 PM

Project Proposal MUST include:

- Technical summary of topic/problem to be studied
- List of primary references to be used/studied
- Role of OpenCL in problem solution

Project Groups

- If you want to work as a group, proposal must describe the work distribution/sharing
  - group topic and work breakdown must be approved by Professor

Map

- Applies a function to every element of a collection of data in parallel
  - **elemental** functions
    - replicated and applied to different data
    - must be *pure* (have no side-effects), to allow all instances of map to execute in any order
      - may read data from memory as long as not modified in parallel
    - each parallel invocation of the elemental function on a different set of data, or portion of the index space, is called an **instance** of the elemental function
  - Map can be thought of as the parallelization of the serial iteration pattern, in the special case that all iterations are independent

- Map is often combined with other patterns
- Map pattern associated with
  - SIMD model : if no control flow in function
  - SPMD model : if control flow in function

Map

- **SAXPY**: $\boldsymbol{y} \leftarrow a\boldsymbol{x} + \boldsymbol{y}$ (Level 1 BLAS routine)
- can be described as a function acting over individual elements, and applying function to every element in input data set

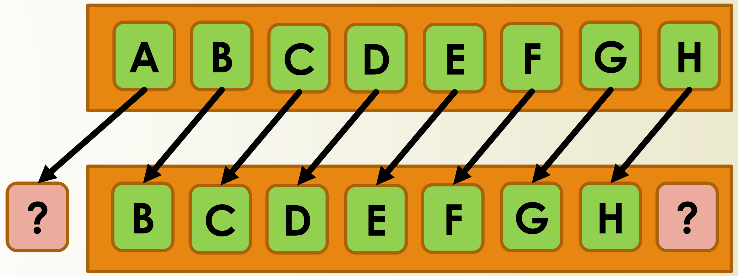
$$f(t,p,q) = tp + q$$
$$\forall i:\ y_i \leftarrow f(a, x_i, y_i)$$

- the **map** pattern invokes the elemental function for every element in input
- elemental functions take two types of arguments:
  - **uniform** parameters : like a, are the same in every invocation of function
  - **varying** parameters : like x_i, y_i, are different for every invocation
- **OpenCL** : kernel functions are equivalent to elemental functions

```
__kernel void saxpy_opencl(__constant float a, __global float* x, __global float* y) {
    int i = get_global_id();
    y[i] = a * x[i] + y[i]; }
```

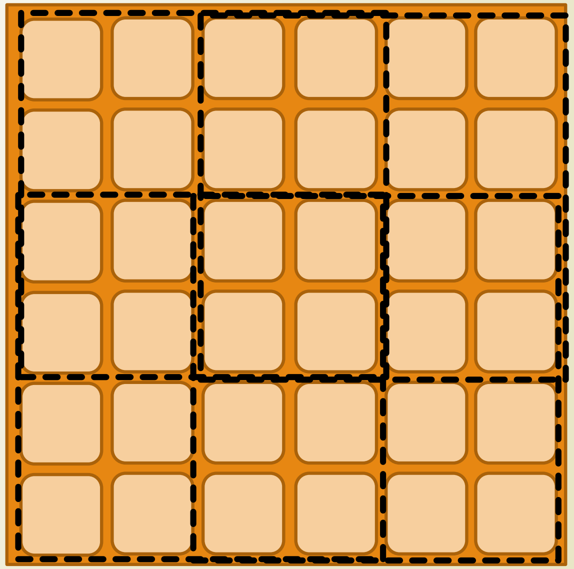
Shift

- A special case of the **gather** pattern
- Moves data left/right/lower/higher using regular pattern of offsets
- Variants based on boundary condition handling:
  - duplicated
  - rotated
  - reflected
  - default value
  - general arbitrary function applied
- Shifts can be multi-dimensional
- Shifts can be efficiently implemented using vector instructions
  - interior data access pattern is regular
- Shift can be used to implement **stencil** pattern



Geometric Decomposition (GD)

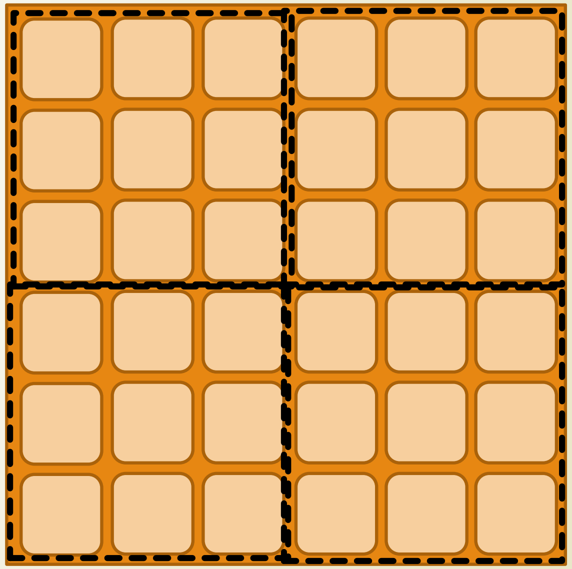
- Applicable when data for a problem can be subdivided following a *divide-and-conquer* strategy.
- When problem has *spatially regular organization*
  - images, regular grids
- Geometric Decomposition: problem subdivision is spatially motivated
- Breaks data into a set of subcollections
  - in general the subcollections can overlap
- Boundary condition issues may arise
  - if input/output domain not evenly divisible
  - if out-of-bound accesses are possible
- GD does not necessarily move data
  - often provides alternate view of data organization



Partition

- A special case of the **GD** pattern
- Data subdivided into *uniform, non-overlapping sections* covering the domain of computation
  - Can be multi-dimensional partition

- **Partition** Properties
  - non-overlapping property avoids write conflicts and race conditions
  - can be applied recursively
  - boundary conditions require special treatment
  - data layout in memory should be considered
  - partitioning is related to the *strip-mining* **stencil** pattern technique
  - often equal-sized regions (to improve load balancing)



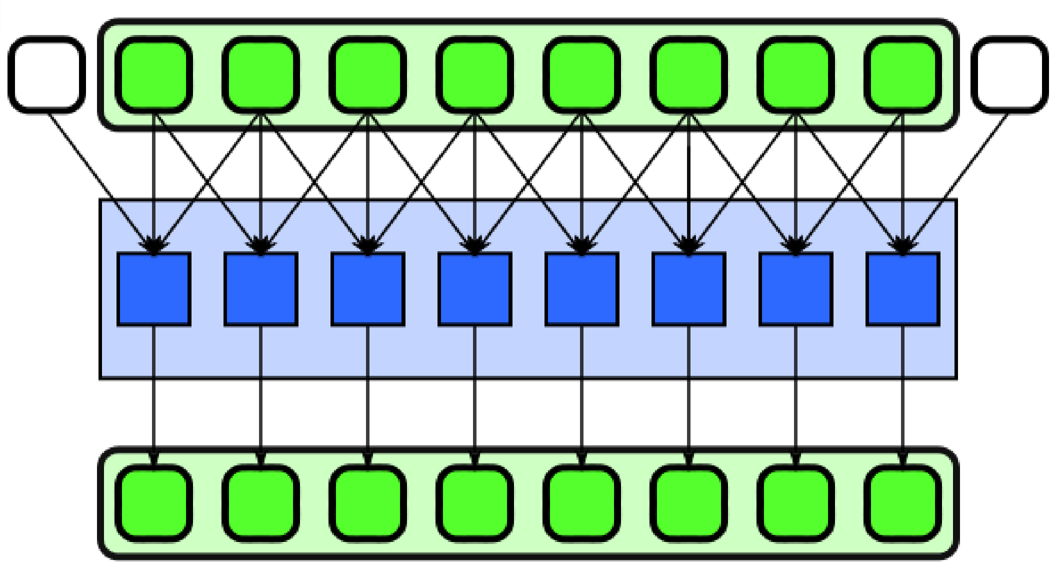
Stencil

- A combination of **map** with a local **gather** over a fixed set of offsets
- Has a regular data access pattern
  - can be implemented using **shifts**.

- Every output of a stencil is a function of some *neighborhood* of elements in the input collection
  - from *square compact* neighborhoods to *sparse* neighborhoods
  - neighborhood structure exposes opportunities for
    - data reuse
    - optimization of data locality

Stencil

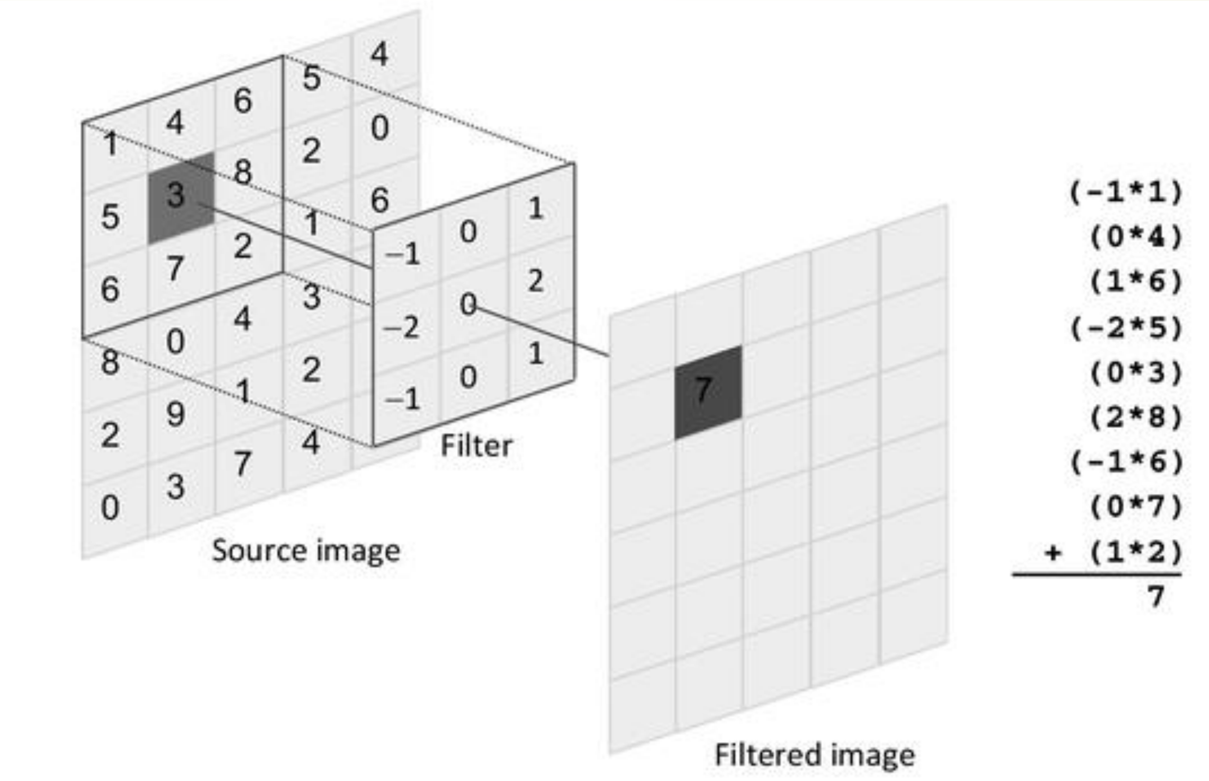
- *Stencil* applies a function to neighbourhoods of a collection.
- Neighbourhoods are given by set of relative offsets.
- Boundary conditions need to be considered, but majority of computation is in interior.



Examples: signal filtering including convolution, median, anisotropic diffusion

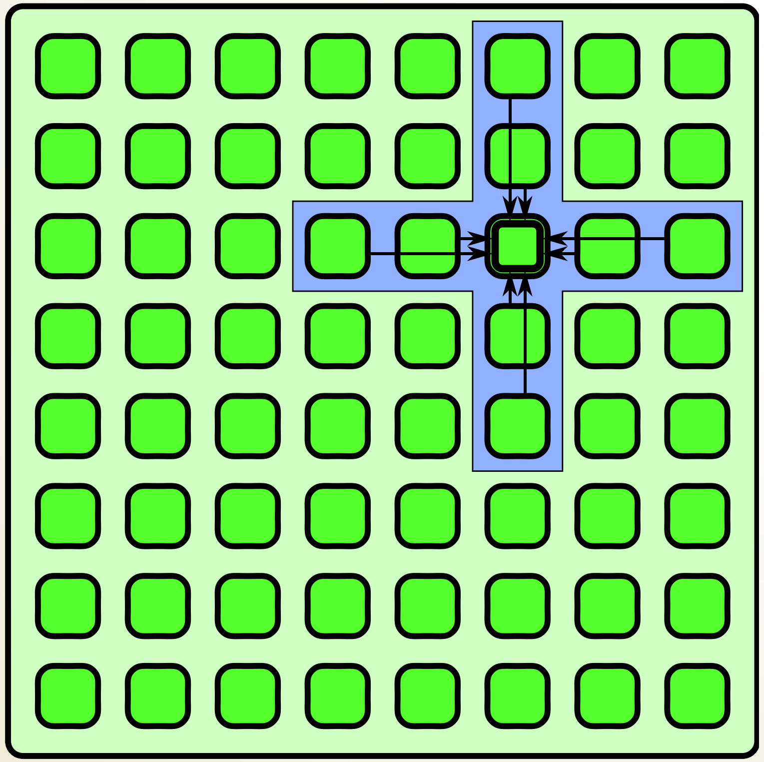
Stencil

- Typically ***input*** divided into a set of partially overlapping strips or regions (a general **GD**) so that neighbors can be accessed.
- ***Output*** is divided into non-overlapping regions (a **partition**) so output can be safely written independently in parallel.



nD Stencil

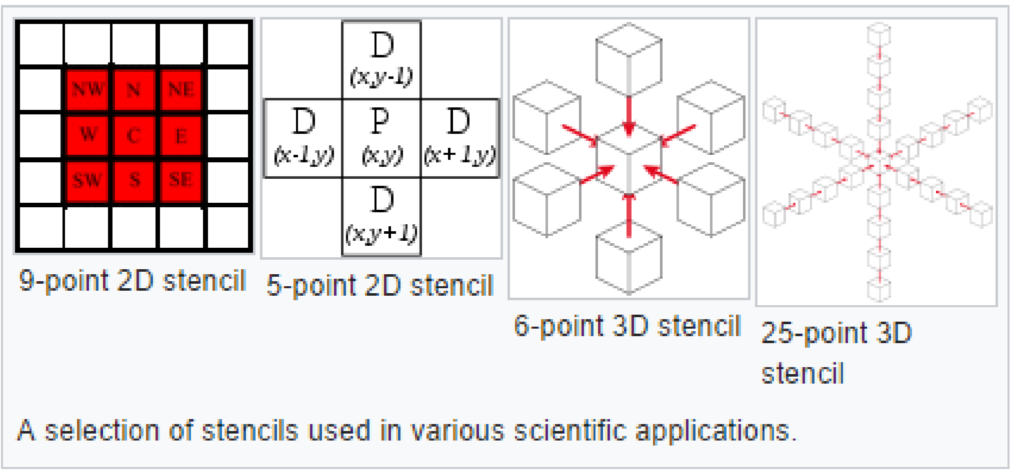
- ***nD Stencil*** applies a function to neighbourhoods of an *nD* array
- *Neighbourhoods* are given by set of relative offsets
- Boundary conditions need to be considered/handled



Examples: image filtering including convolution, median, anisotropic diffusion; simulation including fluid flow, electromagnetic, and financial PDE solvers, lattice QCD

Stencil

- Stencils are extremely common in image processing and scientific simulation
  - Implement finite-difference operators and PDE solvers over regular grids
  - Often 2D or 3D. Can be extended to higher dimensions



[credit: https://en.wikipedia.org/wiki/Stencil_code]

- Some common characteristics of stencil computations
  - High memory traffic
  - Low operational intensity
  - Computations are often memory-bound

Convolution Stencil Operator

Quick Review: General discrete 1D convolution

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \quad \rightarrow \quad (f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

Reminder:

f = [1 1 1 1 1]

g = [1 2 3 2 1]

Stencil

Sequential 1D convolution code

$$(f * g)[n] \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

```
float h_filt[] = {1,2,3,2,1};
int filtLen = 5;
// assume data is in array x with length = dataLen
for(int i = 0; i < dataLen; ++i) {
   y[i]=0;
   for(int j = 0; j < filtLen; ++j) {
      (0 > (i-j)) ? (y[i]+=0) : (y[i] += x[i-j] * h_filt[j]);
   }
}
```

1D convolution stencil kernel

```
__kernel void conv1D(__global float* y, __global float* x,
   int dataLen, __global float* h_filt, int filtLen)
{
    int i = get_global_id(0);
    for(int j = 0; j<filtLen; ++j) {
        (0 > (i-j)) ? (y[i]+=0) : (y[i] += x[i-j] * h_filt[j]); }
}
```

Working with Images in OpenCL

OpenCL Image Objects and the Image API

OpenCL Image Objects and API

- Image objects used to store 1D, 2D, or 3D texture, frame-buffer, or image

Host API

- Creating Image Objects

```
cl_mem clCreateImage (cl_context context, cl_mem_flags flags,
   const cl_image_format *image_format, const cl_image_desc *image_desc,
   void *host_ptr, cl_int *errcode_ret)
```

- Image Format Descriptor structure

```
typedef struct _cl_image_format {
    /* number of channels and channel memory layout */
      cl_channel_order     image_channel_order;
    /* size of the channel data type */
      cl_channel_type      image_channel_data_type;
} cl_image_format;
```

OpenCL Image Objects and API

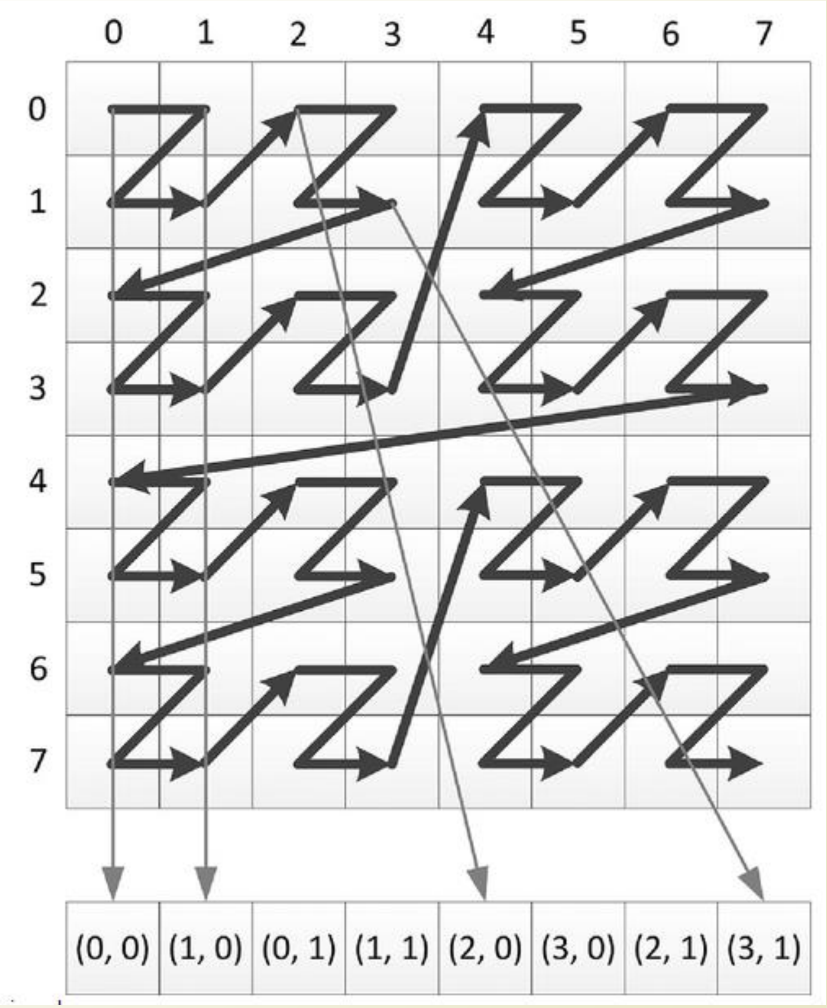
- **Image Descriptor** structure
  - specifies type and dimensions of the image or image array

```
typedef struct cl_image_desc {
    cl_mem_object_type     image_type;
    size_t                 image_width;
    size_t                 image_height;
    size_t                 image_depth;
    size_t                 image_array_size;
    size_t                 image_row_pitch; /* = 0 */
    size_t                 image_slice_pitch; /* = 0 */
    cl_uint                num_mip_levels; /* = 0 */
    cl_uint                num_samples; /* = 0 */
    cl_mem                 mem_object;
} cl_image_desc;
```

image_type must be either
CL_MEM_OBJECT_IMAGE1D, CL_MEM_OBJECT_IMAGE1D_BUFFER, CL_MEM_OBJECT_IMAGE1D_ARRAY, CL_MEM_OBJECT_IMAGE2D, CL_MEM_OBJECT_IMAGE2D_ARRAY or CL_MEM_OBJECT_IMAGE3D

“Opaque” Image objects

- Can transparently use optimized HW memory layouts for 2D data locality
  - such as Z-order (Morton-order)
- Benefit from built-in boundary handling
  - via Samplers
- Allow HW to take advantage of
  - spatial locality
  - built-in HW acceleration
    - linear/bi-linear/tri-linear interpolation
      - via Samplers



OpenCL Image Objects and API

Samplers

- Helper objects which define coordinates, addressing, filtering of an image
  - Coordinate system
    - CLK_NORMALIZED_COORDS_TRUE
    - CLK_NORMALIZED_COORDS_FALSE
  - Addressing mode : defines how out-of-bound image coordinates are handled
    - CLK_ADDRESS_MIRRORED_REPEAT
    - CLK_ADDRESS_REPEAT
    - CLK_ADDRESS_CLAMP_TO_EDGE
    - CLK_ADDRESS_CLAMP (default)
    - CLK_ADDRESS_NONE
  - Filter mode:
    - CLK_FILTER_NEAREST (default)
    - CLK_FILTER_LINEAR

OpenCL Image Objects and API

Samplers

- Can be created in Host code and passed as argument to kernel

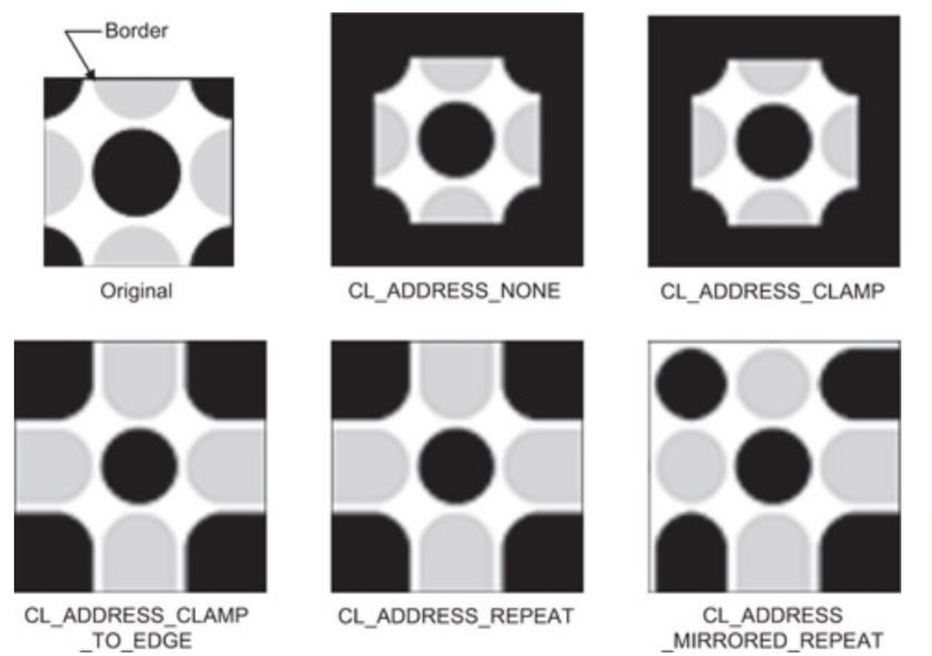
```
cl_sampler clCreateSamplerWithProperties (cl_context context,
    const cl_sampler_properties *sampler_properties,
    cl_int *errcode_ret )

cl_sampler image_sampler_0 = clCreateSampler(context, false,
    CL_ADDRESS_CLAMP_TO_EDGE, CL_FILTER_LINEAR, &err);
```

- Can also be declared in outermost scope of kernel program,
- Can also be declared as global constants in the kernel program source:

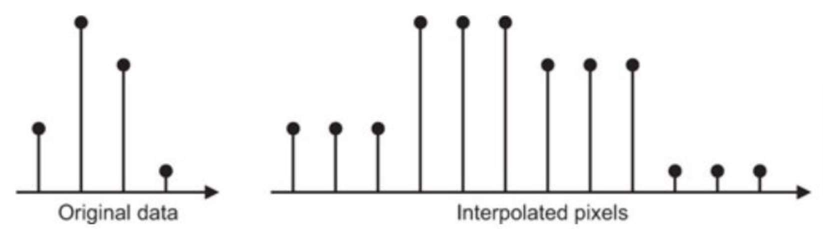
```
const sampler_t samplerA = CLK_NORMALIZED_COORDS_TRUE |
                           CLK_ADDRESS_REPEAT |
                           CLK_FILTER_NEAREST;
```

Specifying Sampler Objects

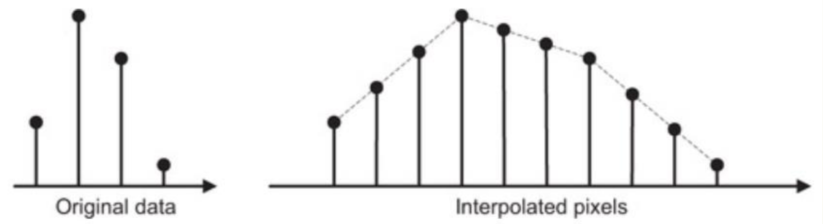
- `cl_addressing_mode`:
  - specifies how out-of-range coordinates are handled
  - what value is returned for a read request to out-of-bound coordinates



- `cl_filter_mode`: interpolation
  - nearest-neighbor



  - linear



OpenCL Image Objects and API

- Enqueuing Image Commands to Read/Write to Device

```
// Write from Host memory to image object on device
cl_int clEnqueueWriteImage (cl_command_queue command_queue,
        cl_mem image,
        cl_bool blocking_write,
        const size_t *origin,
        const size_t *region,
        size_t input_row_pitch,
        size_t input_slice_pitch,
        const void * ptr,
        cl_uint num_events_in_wait_list,
        const cl_event *event_wait_list,
        cl_event *event)

// Read to Host memory from image object on device
cl_int clEnqueueReadImage (cl_command_queue command_queue,
        cl_mem image,
        cl_bool blocking_read,
        const size_t *origin,
        const size_t *region,
        size_t row_pitch,
        size_t slice_pitch,
        void *ptr,
        cl_uint num_events_in_wait_list,
        const cl_event *event_wait_list,
        cl_event *event)
```

- For more Host API details : ***OpenCL API Spec Section 5.3 Image Objects***

OpenCL Image Objects and API

- Mapping Image Region Into Host Address Memory Space

```
void* clEnqueueMapImage(cl_command_queue command_queue,
                        cl_mem image,
                        cl_bool blocking_map,
                        cl_map_flags map_flags,
                        const size_t *origin,
                        const size_t *region,
                        size_t *image_row_pitch,
                        size_t *image_slice_pitch,
                        cl_uint num_events_in_wait_list,
                        const cl_event *event_wait_list,
                        cl_event *event,
                        cl_int *errcode_ret)
```

- And don't forget…

```
err = clEnqueueUnmapMemObject(commands, image_image_IN,
image_image_IN_out, 0, NULL, NULL);
```

- For more Host API details : ***OpenCL API Spec Section 5.3 Image Objects***

OpenCL Image Objects and API

OpenCL C : Device-side kernel Image API

- Image memory objects that are being
  - read : should be declared with `read_only` qualifier

    ```
    float4 read_imagef(read_only image2d_t image, sampler_t sampler, int2 coord )
    float4 read_imagef(read_only image2d_t image, sampler_t sampler, float2 coord )
    float4 read_imagef(aQual image2d_t image, int2 coord )
    ```

  - written to : declare with `write_only` qualifier

    ```
    void write_imagef( aQual image2d_t image, int2 coord, float4 color)
    ```

  - read & written by kernel : declare as `read_write` qualifier
    - probably need `atomic_work_item_fence(CLK_IMAGE_MEM_FENCE)`
      - built-in function to make sure that sampler-less writes are visible to later reads by same work-item.
    - `work_group_barrier(CLK_IMAGE_MEM_FENCE)`
      - Needed if multiple work-items are writing to and reading from multiple locations in an image
- Also refer to: ***OpenCL C Language Specification: section 6.13.14***

stb_image.h : Image Loader Utility

- #include only header-file implementations
- add to top of your source code file

```
#define STB_IMAGE_IMPLEMENTATION
#include " “<your_path>\stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include “<your_path>\stb_image_write.h"
```

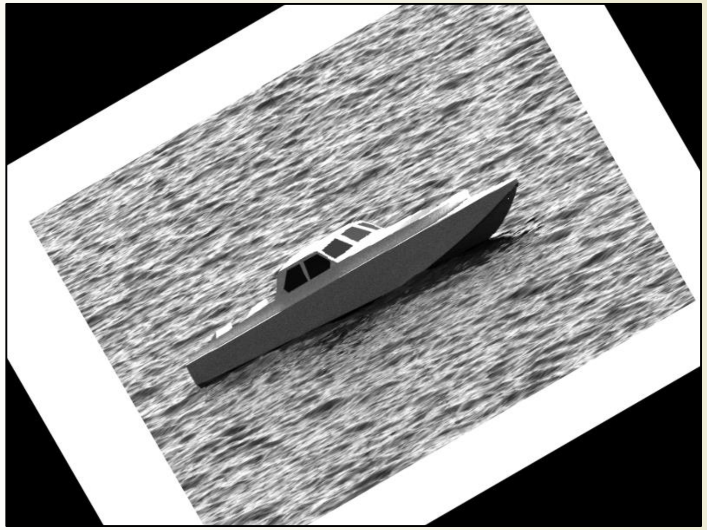
- Convenient functions to load and write to image formats (jpg, png, bmp, …)

```
STBIDEF stbi_uc *stbi_load (char const *filename, int *x, int *y, int *channels_in_file, int desired_channels);
STBIDEF stbi_us *stbi_load_16 (char const *filename, int *x, int *y, int *channels_in_file, int desired_channels);

int stbi_write_png(char const *filename, int w, int h, int comp, const void *data, int stride_in_bytes);
int stbi_write_jpg(char const *filename, int w, int h, int comp, const void *data, int quality);
```

- Visit: https://github.com/nothings/stb

Image Rotation Kernel

Original (Input) Image



Output Image: 30° rotation

- Inputs: original image, rotation angle θ, center point of rotation
  - Input image will be grayscale (monochromatic), single-channel
- Algorithm:
  - coordinates of input point (x,y) when rotated by angle θ around (x0,y0) become (x',y')

$$x' = \cos\theta\,(x - x_0) + \sin\theta\,(y - y_0),$$
$$y' = -\sin\theta\,(x - x_0) + \cos\theta\,(y - y_0).$$

Image Rotation Kernel

- Parallelizing:
  - each output pixel (x',y') can be computed independently
  - use OpenCL support for floating-point coordinates and linear interpolation
  - map global size to image dimensions
    - each WI uses global ID as (x', y')
    - (x0, y0) corresponds to image center
  - Compute locations to read from input image (x, y) using

$$x = x' \cos\theta - y' \sin\theta + x_0,$$
$$y = x' \sin\theta + y' \cos\theta + y_0.$$

  - Built-in OpenCL C functions: read_imagef( ) / write_imagef( )
    - float4 vector data types
  - Sampler object configuration handles coordinates, out-of-bounds access behavior, linear interpolation using fixed-function HW

Rotate Kernel

```
__kernel void img_rotate(__read_only image2d_t inputImg, __write_only image2d_t outputImg, int imgWidth, int imgHeight, float theta)
{
	// use global IDs for output coords
	int x = get_global_id(0);
	int y = get_global_id(1);

	// compute image center
	float x0 = imgWidth/2.0f;
	float y0 = imgHeight/2.0f;

	// compute WI location relative to image center
	int xprime = x-x0;
	int yprime = y-y0;

	// compute sin and cos
	float sinTheta = sin(theta*M_PI_F/180.f);
	float cosTheta = cos(theta*M_PI_F/180.f);

	// compute input location
	float2 readCoord;
	readCoord.x = xprime*cosTheta - yprime*sinTheta + x0;
	readCoord.y = xprime*sinTheta + yprime*cosTheta + y0;

	// read input image
	float value = read_imagef(inputImg, sampler, readCoord).x;  // return only x component of float4 (monochromatic image)

	// write output image
	// write to all R-G-B components, will convert from 32-bit uint to 8-bit uints?
	write_imagef(outputImg, (int2)(x,y), (float4)(value, value, value, 0.f));
}
```

In-class Exercise 6a

- Rotation Kernel using Code Builder KDF
- See <Course website> Files/InclassExercises/EX6

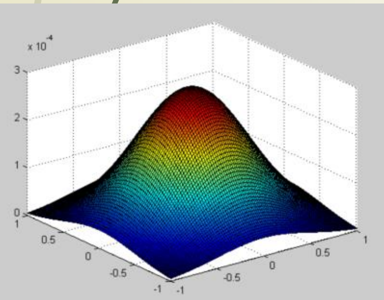
Image Convolution Filter Kernel: Gaussian Blur

Original (Input) Image

Output: 5x5 blur, light

Output: 7x7 blur, moderate

- 2D Gaussian Blur stencil (*blurring, loss of high resolution, low-pass, smoothing*)



$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

pre-computed MxM array of filter coefficients (LUT)

- Input image: Nx-by-Ny (cols, rows) pixel dimensions, grayscale (single-channel)

Image Convolution Filter Kernel: Gaussian Blur

```
/* Iterate over the rows of the source image */
for (int i = 0; i < rows; i++)
{
    /* Iterate over the columns of the source image */
    for (int j = 0; j < cols; j++)
    {
        /* Reset sum for new source pixel */
        int sum = 0;

        /* Apply the filter to the neighborhood */
        for (int k = -halfFilterWidth; k <= halfFilterWidth; k++)
        {
            for (int l = -halfFilterWidth; l <= halfFilterWidth; l++)
            {
                /* Indices used to access the image */
                int r = i+k;
                int c = j+l;

                /* Handle out-of-bounds locations by clamping to
                 * the border pixel */
                r = (r < 0) ? 0 : r;
                c = (c < 0) ? 0 : c;
                r = (r >= rows) ? rows-1 : r;
                c = (c >= cols) ? cols-1 : c;

                sum += Image[r][c] *
                       Filter[k+halfFilterWidth][l+halfFilterWidth];
            }
        }

        /* Write the new pixel value */
        outputImage[i][j] = sum;
    }
}
```

serial 2D convolution

- Parallelization Strategy:
  - Apply MAP+SHIFT+STENCIL+GATHER patterns
  - Use parallelism to remove outer 2 loops
  - Create on WI per output pixel
  - Inner loops provide stencil filter operation
  - Sampler handles out-of-of-bound accesses
  - convolution filter coefficients array stored in OpenCL `__constant` memory

Gaussian Blur Kernel

```
__constant float gaussBlurFilter[25] = {
1.0f/273.0f, 4.0f/273.0f, 7.0f/273.0f, 4.0f/273.0f, 1.0f/273.0f,
4.0f/273.0f, 16.0f/273.0f, 26.0f/273.0f, 16.0f/273.0f, 4.0f/273.0f,
7.0f/273.0f, 26.0f/273.0f, 41.0f/273.0f, 26.0f/273.0f, 7.0f/273.0f,
4.0f/273.0f, 16.0f/273.0f, 26.0f/273.0f, 16.0f/273.0f, 4.0f/273.0f,
1.0f/273.0f, 4.0f/273.0f, 7.0f/273.0f, 4.0f/273.0f, 1.0f/273.0f
};

__constant int filterWidth = 5;

__kernel void img_conv_filter(__read_only image2d_t inputImg, __write_only image2d_t outputImg, int cols, int rows)
{
        // use global IDs for output coords
        int x = get_global_id(0); // columns
        int y = get_global_id(1); // rows

        int halfWidth = (int)(filterWidth/2);    // auto-round nearest int ???

        float4 sum = (float4)(0);

        int filtIdx = 0;  // filter kernel passed in as linearized buffer array
        int2 coords;

        for(int i = -halfWidth; i <= halfWidth; i++) // iterate filter rows
        {
        coords.y = y + i;
        for(int j = -halfWidth; j <= halfWidth; j++) // iterate filter cols
        {
        coords.x = x + j;
        //float4 pixel = convert_float4(read_imageui(inputImg, sampler, coords)); // operate element-wise on all 3 color components (r,g,b)
        float4 pixel = read_imagef(inputImg, sampler, coords); // operate element-wise on all 3 color components (r,g,b)
        filtIdx++;
        sum += pixel * (float4)(gaussBlurFilter[filtIdx],gaussBlurFilter[filtIdx],gaussBlurFilter[filtIdx],1.0f); // leave a-channel unchanged
        }

        }

        //write resultant filtered pixel to output image
        coords = (int2)(x,y);
        //write_imageui(outputImg, coords, convert_uint4(sum));
        write_imagef(outputImg, coords, sum);
}
```

In-class Exercise 6b

- Gaussian Blur Kernels using Code Builder KDF
- See <Course website> Files/InclassExercises/EX6

HW4 is posted

- DUE Thursday, Nov 15 by 6:00 PM